Declaring Variables in Assembly Language

As in Java, variables must be declared before they can be used. Unlike Java, we do not specify a variable **type** in the declaration in assembly language. Instead we declare the name and **size** of the variable, i.e. the number of bytes the variable will occupy. We may also specify an initial value.

A **directive** (i.e. a command to the assembler) is used to define variables. In 8086 assembly language, the directive **`db`** defines a byte sized variable; **`dw`** defines a word sized variable (16 bits) and **`dd`** defines a double word (long word, 32 bits) variable.

A Java variable of type **`int`** may be implemented using a size of 16 or 32 bits, i.e. **`dw`** or **`dd`** is used. A Java variable of type `char`, which is used to store a single character, is implemented using the **`db`** directive.

Example:

```
reply db ‘y’
prompt db ‘Enter your favourite colour: ’, 0
colour db 80 dup(?)
i db 20
k db ?
num dw 4000
large dd 50000
```

`reply` is defined as a character variable, which is initialised to

‘y’.

prompt is defined as a string, terminated by the Null character.

The definition of the variable colour demonstrates how to declare an **array** of characters of size 80, which contains undefined values.

The purpose of dup is to tell the assembler to duplicate or repeat the data definition directive a specific number of times, in this case 80 dup specifies that 80 bytes of storage are to be set aside since dup is used with the db directive.

The (?) with the dup means that storage allocated by the directive is unitialised or undefined.

i and k are byte sized variables, where i is initialised to 20 and k is left undefined.

num is a 16-bit variable, initialised to 4000 and the variable large is a 32-bit variable, initialised to 15000.

Indirect Addressing

Given that we have defined a string variable **message** as

```
message db ‘Hello’,0,
```

an important feature is that the characters are **stored in consecutive memory locations**.

If the ‘H’ is in location 1024, then ‘e’ will be in location 1025, ‘l’ will be in location 1026 and so on. A technique known as

indirect addressing may be used to access the elements of the array.

Indirect addressing allows us store the address of a location in a register and use this register to access the value stored at that location.

This means that we can store the address of the string in a register and access the first character of the string via the register. If we increment the register contents by 1, we can access the next character of the string. By continuing to increment the register, we can access each character of the string, in turn, processing it as we see fit.

Figure 1 illustrates how indirect addressing operates, using register `bx` to contain the address of a string `"Hello"` in memory. Here, register `bx` has the value `1024` which is the address of the first character in the string.

Another way of phrasing this is to say that `bx` **points** to the first character in the string.

In 8086 assembly language we denote this by enclosing `bx` in square brackets: **`[bx]`**, which reads as the value **pointed to** by `bx`, i.e. **the contents of the location whose address is stored in the `bx` register**.

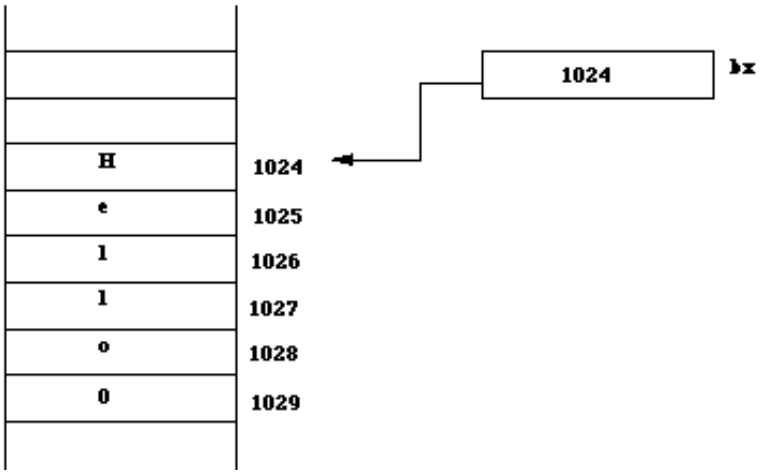

Figure 1: Using the `bx` register for indirect addressing

The first character of the string can be accessed as follows:

```
cmp byte ptr [bx], 0 ; is this end of string?
```

This instruction compares the character (indicated by `byte ptr`) pointed to by `bx` with 0.

How do we store the address of the string in `bx` in the first place? The special operator **`offset`** allows us specify the address of a memory variable. For example, the instruction:

```
mov bx, offset message
```

will store the **address** of the variable `message` in `bx`. We can then use `bx` to access the variable `message`.

Example: The following code fragment illustrates the use of indirect addressing. It is a loop to count the number of characters in a string terminated by the Null character (ASCII 0). It uses the `cx` register to store the number of characters in the string.

```
message db ‘Hello’, 0

.......
```

```
........
mov cx, 0 ; cx stores number of characters
mov bx, offset message ; store address of message in bx
begin:
cmp byte ptr [bx], 0 ; is this end of string?
je fin ; if yes goto Finished
inc cx ; cx = cx + 1
inc bx ; bx points to next character
jmp begin
; cx now contains the # of
; characters in message
fin:
```

The label `begin` indicates the beginning of the loop to count the characters. After executing the `mov` instruction, register `bx` contains the address of the first character in the string. We compare this value with `0` and if the value is not `0`, we count it by incrementing `cx`. We then increment `bx` so that it now points to the next character in the string. We repeat this process until we reach the `0` character which terminates the string.

Note: If you omit the 0 character when defining the string, the above program will fail. Why? The reason is that the loop continues to execute, until `bx` points to a memory location containing 0. If 0 has been omitted from the definition of message, then we do not know when, if ever, the loop will terminate. This is the same as an array subscript out of bounds error in a high level language.

The form of indirect addressing described here is called **register indirect addressing** because a register is used store the indirect address.

String I/O

In programming languages such as C, strings are terminated by the `'\0'` character. We adopt the same convention. This method of terminating a string has an advantage over that used for the `puts` subprogram defined earlier, where the '$' character is used to terminate a string. The use of the value 0 to terminate a string means that a string may contain the '$' character which can then be displayed, since '$' cannot be displayed by `puts`.

We use this indirect addressing in the implementation of two subprograms for reading and displaying strings: `get_str` and `put_str`

Example 3.42: Read colour entered by the user and display a suitable message, using `get_str` and `put_str`.

```
; colour.asm: Prompt user to enter a colour and display a message
; Author: Joe Carthy
; Date: March 1994
.model small
.stack 256
CR equ 13d
LF equ 10d
; string definitions: note 0 terminator
.data
msg1 db 'Enter your favourite colour: ', 0
```

```
msg2 db CR, LF,‘Yuk ! I hate ‘, 0
colour db 80 dup (0)
.code
start:
mov ax, @data
mov ds, ax
mov ax, offset msg1
call put_str ; display prompt
mov ax, offset colour
call get_str ; read colour
mov ax, offset msg2
call put_str ; display msg2
mov ax, offset colour
call put_str ; display colour entered by user
mov ax, 4c00h
int 21h ; finished, back to dos
```

```
put_str: ; display string terminated by 0
; whose address is in ax
push ax ; save registers
push bx
push cx
```

```
push dx
mov bx, ax ; store address in bx
mov al, byte ptr [bx] ; al = first char in string
put_loop: cmp al, 0 ; al == 0 ?
je put_fin ; while al != 0
call putc ; display character
inc bx ; bx = bx + 1
mov al, byte ptr [bx] ; al = next char in string
jmp put_loop ; repeat loop test
put_fin:
pop dx ; restore registers
pop cx
pop bx
pop ax
ret



get_str: ; read string terminated by CR into array
; whose address is in ax
push ax ; save registers
push bx
push cx
push dx
mov bx, ax
call getc ; read first character
mov byte ptr [bx], al ; In C: str[i] = al
get_loop: cmp al, 13 ; al == CR ?
```

```
je get_fin ;while al != CR
inc bx ; bx = bx + 1
call getc ; read next character
mov byte ptr [bx], al ; In C: str[i] = al
jmp get_loop ; repeat loop test
get_fin: mov byte ptr [bx], 0 ; terminate string with 0
pop dx ; restore registers
pop cx
pop bx
pop ax
ret


putc: ; display character in al
push ax ; save ax
push bx ; save bx
push cx ; save cx
push dx ; save dx
mov dl, al
mov ah, 2h
int 21h
pop dx ; restore dx
pop cx ; restore cx
pop bx ; restore bx
pop ax ; restore ax
ret
getc: ; read character into al
push bx ; save bx
```

```
push cx ; save cx
push dx ; save dx
mov ah, 1h
int 21h
pop dx ; restore dx
pop cx ; restore cx
pop bx ; restore bx
ret
end start
```

This program produces as output:

```
Enter your favourite colour: yellow
Yuk ! I hate yellow
```

Reading and Displaying Numbers

See Chapter 3 of textbook for implementation details

We use `getn` and `putn` to read and display numbers:

`getn:` reads a number from the keyboard and returns it in the ax register

`putn:` displays the number in the ax register

Example: Write a program to read two numbers, add them and display the result.

```
; calc.asm: Read and sum two numbers. Display result.
```

```
; Author: Joe Carthy
; Date: March 1994
.model small
.stack 256
CR equ 13d
LF equ 10d
.data
prompt1 db ‘Enter first number: ’, 0
prompt2 db CR, LF,‘Enter second number:’,0
result db CR, LF ‘The sum is’, 0
num1 dw ?
num2 dw ?

.code
start:
mov ax, @data
mov ds, ax
mov ax, offset prompt1
call put_str ; display prompt1
call getn ; read first number
mov num1, ax
mov ax, offset prompt2
call put_str ; display prompt2
call getn ; read second number
mov num2, ax
mov ax, offset result
call put_str ; display result message
mov ax, num1 ; ax = num1
```

```
add ax, num2 ; ax = ax + num2
call putn ; display sum
mov ax, 4c00h
int 21h ; finished, back to dos


<definitions of getn, putn, put_str, get_str, getc, putc go here>
end start
```

Running the above program produces:

```
Enter first number: 8
Enter second number: 6
The sum is 14
```

More about the Stack

A **stack** is an **area of memory** which is used for storing data on a temporary basis. In a typical computer system the memory is logically partitioned into separate areas. Your program code is stored in one such area, your variables may be in another such area and another area is used for the stack. Figure 2 is a crude illustration of how memory might be allocated to a user program running.

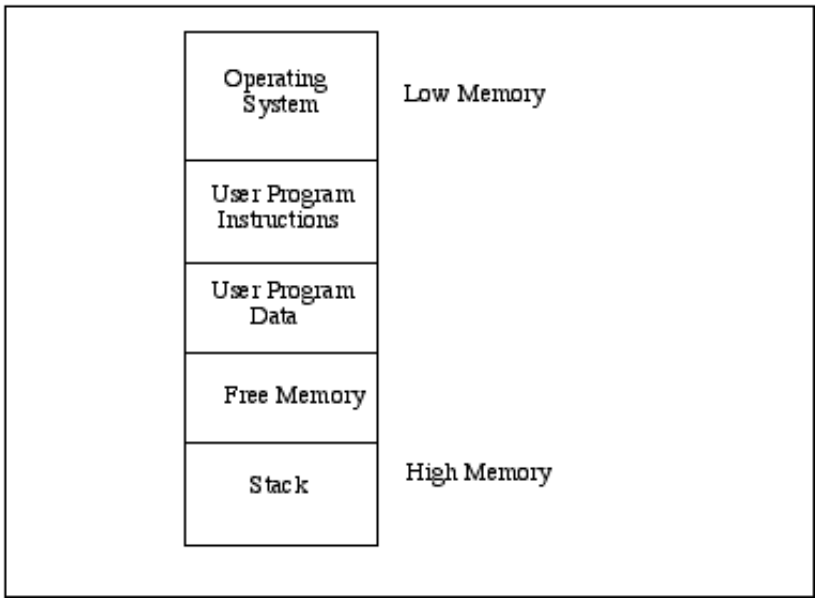

Figure 2: Memory allocation: User programs share memory with the Operating System software

The area of memory with addresses near 0 is called low memory, while high memory refers to the area of memory near the highest address. The area of memory used for your program code is fixed, i.e. once the code is loaded into memory it does not grow or shrink.

The stack on the other hand may require varying amounts of memory. The amount actually required depends on how the program uses the stack. Thus the size of the stack varies during program execution. We can store information on the stack and retrieve it later.

One of the most common uses of the stack is in the implementation of the subprogram facility. This usage is transparent to the programmer, i.e. the programmer does not have to explicitly access the stack. The instructions to call a subprogram and to return from a subprogram automatically access the stack. They do this in order to return to the correct place in your program when the subprogram is finished.

The point in your program where control returns after a

subprogram finishes is called the **return address.** The return address of a subprogram is placed on the stack by the `call` instruction. When the subprogram finishes, the `ret` instruction retrieves the return address from the stack and transfers control to that location. The stack may also be used to pass information to subprograms and to return information from subprograms, i.e. as a mechanism for handling high level language parameters.

Conceptually a stack as its name implies is a **stack of data elements**. The size of the elements depends on the processor and for example, may be 1 byte, 2 bytes or 4 bytes. We will ignore this for the moment. We can illustrate a stack as in Figure 3:

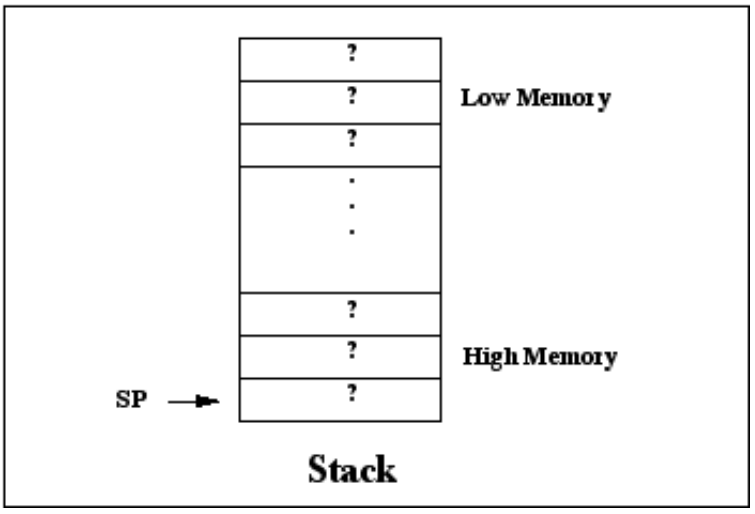

Figure 3: Simple model of the stack

To use the stack, the processor must keep track of where items are stored on it. It does this by using the **stack pointer** (**`sp`**) register.

This is one of the processor's special registers. It points to the **top** of the stack, i.e. its contains the address of the stack memory element containing the value last placed on the stack. When we place an element on the stack, the stack pointer contains the address of that element on the stack. If we place a number of elements on the stack, the stack pointer will always point to the last

element we placed on the stack. When retrieving elements from the stack we retrieve them in reverse order. This will become clearer when we write some stack manipulation programs.

There are two basic stack operations which are used to manipulate the stack usually called **push** and **pop**. The 8086 `push` instruction places (pushes) a value on the stack. The stack pointer is left pointing at the value pushed on the stack. For example, if `ax` contains the number `123`, then the following instruction:

```
push ax
```

will cause the value of `ax` to be stored on the stack. In this case the number `123` is stored on the stack and `sp` **points** to the location on the stack where `123` is stored.

The 8086 `pop` instruction is used to retrieve a value previously placed on the stack. The stack pointer is left pointing at the next element on the stack. Thus `pop` conceptually removes the value from the stack. Having stored a value on the stack as above, we can retrieve it by:

```
pop ax
```

which transfers the data from the top of the stack to `ax`, (or any register) in this case the number `123` is transferred. Information is stored on the stack starting from high memory locations. As we place data on the stack, the stack pointer points to successively lower memory locations. We say that the stack grows downwards. If we assume that the top of the stack is location `1000` (`sp` contains `1000`) then the operation of `push ax` is as follows.

Firstly, `sp` is decremented by the size of the element (2 bytes for the 8086) to be pushed on the stack. Then the value of `ax` is copied to the location pointed to by `sp`, i.e. `998` in this case. If we then

assign bx the value 212 and carry out a push bx operation, sp is again decremented by two, giving it the value 996 and 212 is stored at this location on the stack. We now have two values on the stack.

As mentioned earlier, if we now retrieve these values, we encounter the fundamental feature of any stack mechanism. Values are retrieved in **reverse order.** This means that the last item placed on the stack, is the first item to be retrieved. We call such a process a **Last-In-First-Out** process or a **LIFO** process.

So, if we now carry out a pop ax operation, ax gets as its value 212, i.e. the last value pushed on the stack.

If we now carry out a pop bx operation, bx gets as its value 123, the second last value pushed on the stack.

Hence, the operation of pop is to copy a value from the top of the stack, as pointed to by sp and to increment sp by 2 so that it now points to the previous value on the stack.

We can push the value of any register or memory variable on the stack. We can retrieve a value from the stack and store it in any register or a memory variable.

The above example is illustrated in Figure 4 (steps (1) to (4) correspond to the states of the stack and stack pointer after each instruction).

Note: For the 8086, we can **only push 16-bit items** onto the stack e.g. any register.

The following are ILLEGAL: push al

pop bh

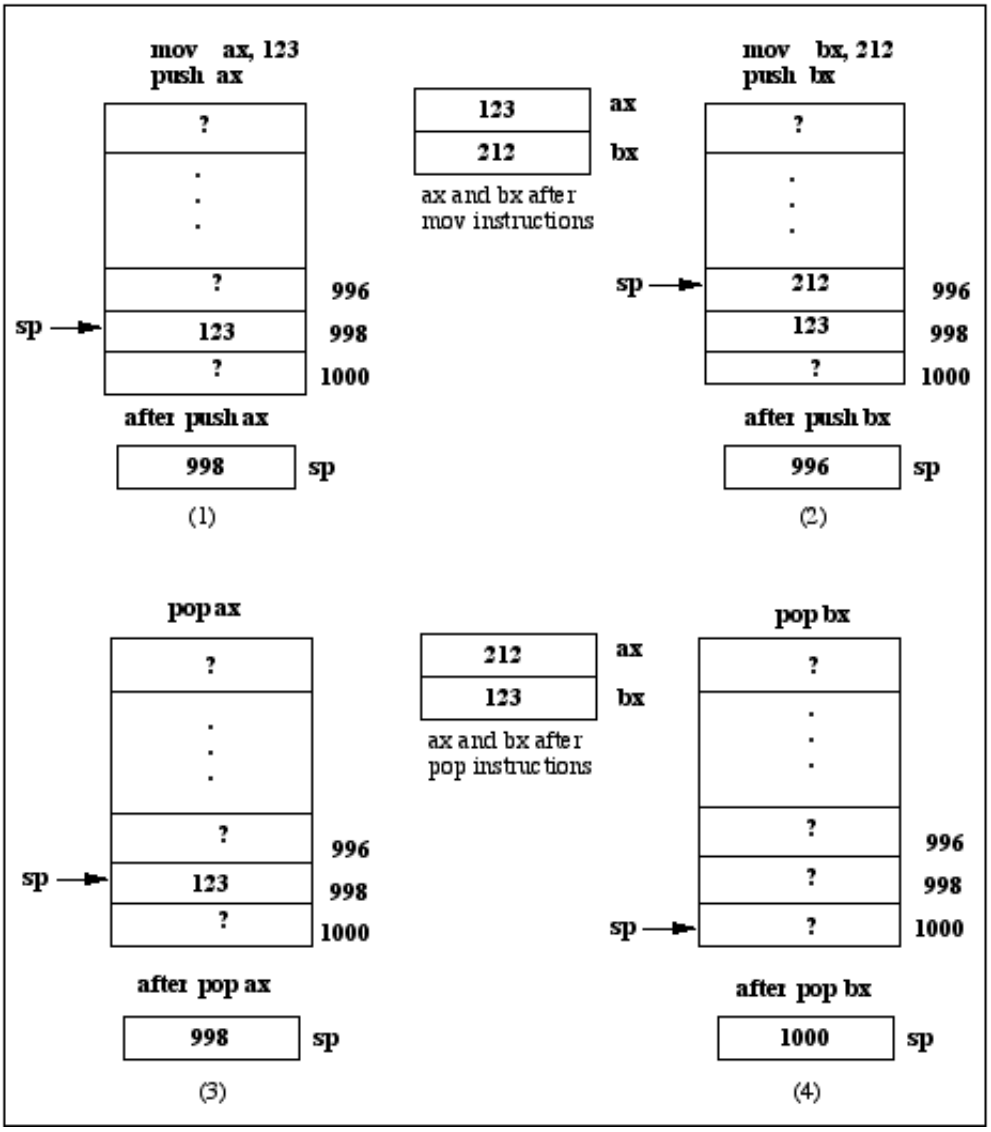

Figure 4: LIFO nature of push and pop

Example: Using the stack, swap the values of the ax and bx registers, so that ax now contains what bx contained and bx contains what ax contained. (This is not the most efficient way to exchange the contents of two variables). To carry out this operation, we need at least one temporary variable:

Version 1**:**

```
push ax ; Store ax on stack
push bx ; Store bx on stack
pop ax ; Copy last value on stack to ax
pop bx ; Copy first value to bx
```

The above solution stores both ax and bx on the stack and utilises the LIFO nature of the stack to retrieve the values in reverse order, thus swapping them in this example. We really only need to store one of the values on the stack, so the following is a more efficient solution.

Version 2:

```
push ax ; Store ax on stack
mov ax, bx ; Copy bx to ax
pop bx ; Copy old ax from stack
```

When using the stack, the number of items pushed on should equal the number of items popped off.

This is vital if the stack is being used inside a subprogram. This is because, when a subprogram is called its return address is pushed on the stack.

If, inside the subprogram, you push something on the stack and do not remove it, the return instruction will retrieve the item you left on the stack instead of the return address. This means that your subprogram cannot return to where it was called from and it will most likely crash (unless you were very clever about what you left on the stack!).

Format of Assembly Language Instructions

The format of assembly language instructions is relatively standard. The **general format** of an instruction is (where square brackets **[]** indicate the **optional** fields) as follows:

```
[Label] Operation [Operands] [; Comment]
```

The instruction may be treated as being composed of four **fields**. All four fields need **not** be present in every instruction, as we have seen from the examples already presented. Unless there is only a comment field, the **operation field** is **always** necessary. The label and the operand fields may or may not be required depending on the operation field.

Example: Examples of instructions with varying numbers of fields.

Note

`L1: cmp bx, cx ; Compare bx with cx` ***all fields present***

`add ax, 25` ***operation and 2 operands***

`inc bx` ***operation and 1 operand***

`ret` ***operation field only***

`; Comment: whatever you wish !!` ***comment field only***

Bit Manipulation

One of the features of assembly language programming is that you can access the individual bits of a byte (word or long word).

You can **set** bits (give them a value of `1`), **clear** them (give them a

value of `0`), **complement** them (change `0` to `1` or `1` to `0`), and **test** if they have a particular value.

These operations are essential when writing subprograms to control devices such as printers, plotters and disk drives. Subprograms that control devices are often called **device drivers**. In such subprograms, it is often necessary to set particular bits in a register associated with the device, in order to operate the device. The instructions to operate on bits are called **logical** instructions.

Under normal circumstances programmers rarely need concern themselves with bit operations. In fact most high-level languages do not provide bit manipulation operations. (The C language is a notable exception). Another reason for manipulating bits is to make programs more efficient. By this we usually mean one of two things: the program is smaller in size and so requires less RAM or the program runs faster.

The Logical Instructions: `and, or, xor, not`

As stated above, the logical instructions allow us operate on the bits of an operand. The operand may be a byte (8 bits), a word (16 bits) a long word (32 bits). We will concentrate on byte sized operands, but the instructions operate on word operands in exactly the same fashion.

Clearing Bits: `and` instruction

A bit **and** operation compares two bits and sets the result to `0` if either of the bits is `0`.

e.g.

```
1 and 0 returns 0

0 and 1 returns 0

0 and 0 returns 0

1 and 1 returns 1
```

The `and` instruction carries out the `and` operation on all of the bits of the source operand with all of the bits of the destination operand, storing the result in the destination operand (like the arithmetic instructions such as `add` and `sub`).

The operation `0 and x` always results in `0` regardless of the value of `x` (`1` or `0`). This means that we can use the `and` instruction to clear a specified bit or collection of bits in an operand.

If we wish to clear, say bit 5, of an 8-bit operand, we `and` the operand with the value `1101 1111`, i.e. a value with bit `5` set to `0` and all other values set to `1`.

This results in bit `5` of the 8-bit operand being cleared, with the other bits remaining unchanged, since `1 and x` always yields `x`.

(Remember, when referring to a bit number, we count from bit **`0`** upwards.)

Example 4.1: To clear bit `5` of a byte we `and` the byte with `1101 1111`

```
mov al, 62h ; al = 0110 0010

and al, 0dfh ; and it with 1101 1111

; al is 42h 0100 0010
```

[Note: You can use binary numbers directly in 8086 assembly

language, e.g.

```
mov al, 01100010b
and al, 11011111b
```

but it is easier to write them using their hexadecimal equivalents.]

The value in the source operand, `0dfh`, in this example, is called a **bit mask**. It specifies the bits in the destination operand that are to be changed. Using the `and` instruction, any bit in the bit mask with value `0` will cause the corresponding bit in the destination operand to be cleared.

In the ASCII codes of the lowercase letters, bit `5` is always `1`. The corresponding ASCII codes of the uppercase letters are identical except that bit `5` is always `0`. Thus to convert a lowercase letter to uppercase we simply need to clear bit `5` (i.e. set bit `5` to `0`). This can be done using the `and` instruction and an appropriate bit mask, i.e. `0dfh`, as shown in the above example. The letter `'b'` has ASCII code `62h`. We could rewrite Example 4.1 above as:

Example B.1: Converting a lowercase letter to its uppercase equivalent:

```
mov al, 'b' ; al = 'b'(= 98d or 62h) 0110 0010
and al, 0dfh ; mask = 1101 1111
; al now = 'B'(= 66d or 42h) 0100 0010
```

The bit mask `1101 1111` when used with `and` will always set bit `5` to `0` leaving the remaining bits unchanged as illustrated below:

```
xxxx xxxx ; destination bits
and 1101 1111 ; and with mask bits
xx0x xxxx ; result is that bit 5 is cleared
```

If the destination operand contains a lowercase letter, the result will be the corresponding uppercase equivalent. In effect, we have subtracted `32` from the ASCII code of the lowercase letter which was the method we used in Chapter 3 for converting lowercase letters to their uppercase equivalents.

Setting Bits: `or` instruction

A bit `or` operation compares two bits and sets the result to `1` if either bit is set to `1`.

e.g.

```
1 or 0 returns 1

0 or 1 returns 1

1 or 1 returns 1

0 or 0 returns 0
```

The `or` instruction carries out an `or` operation with all of the bits of the source and destination operands and stores the result in the destination operand.

The `or` instruction can be used to set bits to `1` regardless of their current setting since `x or 1` returns `1` regardless of the value of `x` (`0` or `1`).

The bits set using the `or` instruction are said to be **masked in**.

Example: Take the conversion of an uppercase letter to lowercase, the opposite of Example B.1 discussed above. Here, we need to **set** bit `5` of the uppercase letter's ASCII code to `1` so that it becomes lowercase and leave all other bits unchanged. The required mask is

`0010 0000` (`20h`). If we store `'A'` in `al` then it can be converted to `'a'` as follows:

```
mov al, 'A' ; al = 'A' = 0100 0001
or al, 20h ; or with 0010 0000
; gives al = 'a' 0110 0001
```

In effect, we have added `32` to the uppercase ASCII code thus obtaining the lowercase ASCII code.

Before changing the case of a letter, it is important to verify that you have a letter in the variable you are working with.

Exercises

4.1 Specify the instructions and masks would you use to

a) set bits `2`, `3` and `4` of the `ax` register

b) clear bits `4` and `7` of the `bx` register

4.2 How would `al` be affected by the following instructions:

(a) `and al, 00fh`

(b) `and al, 0f0h`

(c) `or al, 00fh`

(d) `or al, 0f0h`

4.3 Write subprograms `todigit` and `tocharacter`, which convert a digit to its equivalent ASCII character code and vice versa.

4.1.3 The `xor` instruction

The xor operation compares two bits and sets the result to 1 if the bits are different.

e.g.

```
1 xor 0 returns 1

0 xor 1 returns 1

1 xor 1 returns 0

0 xor 0 returns 0
```

The xor instruction carries out the xor operation with its operands, storing the result in the destination operand.

The xor instruction can be used to **toggle** the value of specific bits (reverse them from their current settings). The bit mask to toggle particular bits should have 1's for any bit position you wish to toggle and 0's for bits which are to remain unchanged.

Example 4.7: Toggle bits 0, 1 and 6 of the value in al (here 67h):

```
mov al, 67h ; al = 0011 0111

xor al, 08h ; xor it with 0100 0011

; al is 34h 0111 0100
```

A common use of xor is to clear a register, i.e. set all bits to 0, for example, we can clear register cx as follows

```
xor cx, cx
```

This is because when the identical operands are xored, each bit cancels itself, producing 0:

```
0 xor 0 produces 0
```

```
1 xor 1 produces 0
```

Thus `abcdefgh xor abcdefgh` produces `00000000` where `abcdefgh` represents some bit pattern. The more obvious way of clearing a register is to use a `mov` instruction as in:

```
mov cx, 0
```

but this is slower to execute and occupies more memory than the `xor` instruction. This is because bit manipulation instructions, such as `xor`, can be implemented very efficiently in hardware. The `sub` instruction may also be used to clear a register:

```
sub cx, cx
```

It is also smaller and faster than the `mov` version, but not as fast as the `xor` version. My own preference is to use the clearer version, i.e. the `mov` instruction. However, in practice, assembly language programs are used where efficiency is important and so clearing a register with `xor` is often used.

4.1.4 The `not` instruction

The `not` operation **complements** or **inverts** a bit, i.e.

```
not 1 returns 0
not 0 returns 1
```

The **`not`** instruction inverts **all** of the bits of its operand.

Example 4.8: Complementing the `al` register:

```
mov al, 33h ; al = 00110011
```

```
not al ; al = 11001100
```

Table 1 summarises the results of the logical operations. Such a table is called a **truth table**.

A	B	not A	A and B	A or B	A xor B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Table 4.1: Truth table for logical operators

Efficiency

As noted earlier, the `xor` instruction is often used to clear an operand because of its efficiency. For similar reasons of efficiency, the `or/and` instructions may be used to compare an operand to `0`.

Example 4.9: Comparing an operand to `0` using logical instructions:

```
or cx, cx ; compares cx with 0
je label
and ax, ax ; compares ax with 0
jg label2
```

Doing `or/and` operations on identical operands, does not change the destination operand (`x or x` returns `x`; `x and x` returns `x`), but they do set flags in the status register. The `or/and` instructions above have the same effect as the `cmp` instructions used in Example 4.10, but they are faster and smaller instructions (each occupies `2` bytes) than the `cmp` instruction (which occupies `3` bytes).

Shifting and Rotating Bits

We sometimes wish to change the positions of all the bits in a byte, word or long word. The 8086 provides a complete set of instructions for shifting and rotating bits. Bits can be moved right (towards the `0` bit) or left towards the most significant bit. Values shifted off the end of an operand are lost (one may go into the **carry flag**).

Shift instructions move bits a specified number of places to the right or left.

Rotate instructions move bits a specified number of places to the right or left. For each bit rotated, the last bit in the direction of the rotate is moved into the first bit position at the other end of the operand.